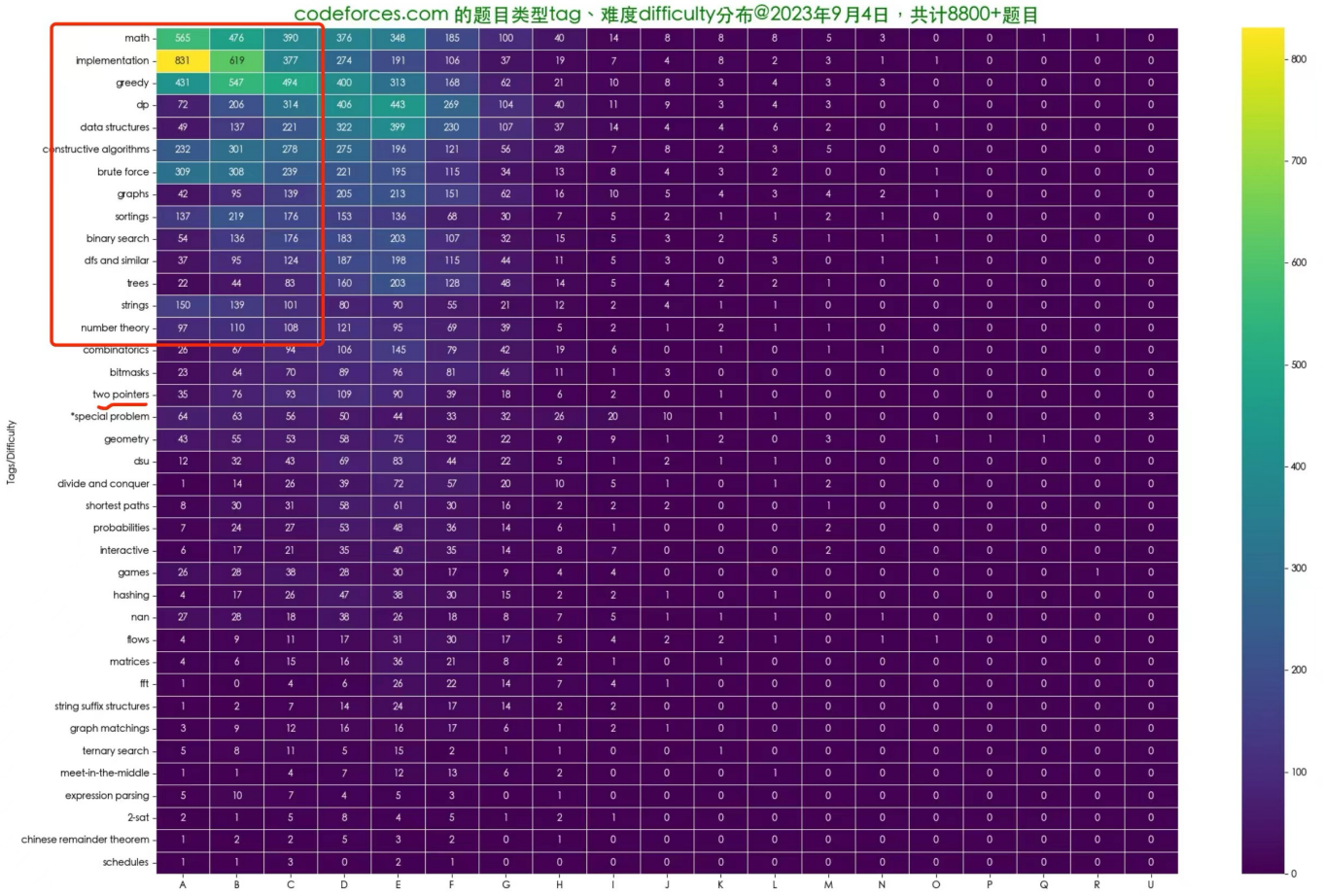
二、CF Tags 题目类型



说明：常用解题步骤/技巧/算法，在codeforces.com中记为Tag，功能是帮助用户找到一类特定的问题。下面是一些常见的标记及其解释：

1. dp (Dynamic Programming): 这是一种算法设计策略，主要用于需要考虑预计结果和当前情境的复杂问题；它将问题分解为更小的子问题，并存储这些子问题的解，以避免重复计算。
2. greedy: 贪婪算法是一种寻找工作最优解的算法，它总是选取当前状态下最好的选项，不考虑结果对未来的影响。
3. math: 这个标签用于那些需要一定数学知识来解答的问题，如概率、组合数学、数论或几何等。
4. ds (Data Structures): 对于需要使用到数据结构（例如数组、链表、队列、栈、哈希表、树、图等）的题目，会使用这个标签。
5. graphs: 这个标签用于图论相关的题目，如最短路径、最小生成树、网络流等问题。
6. sortings：这个标签用于涉及各种排序方法（比如冒泡排序，选择排序，插入排序，快速排序等）的题目。
7. binary search: 对于涉及到二分查找的题目，会使用这个标签。
8. constructive algorithms: 这个标签用于那些需要构造性地找出解的问题，即不仅要找出一个解，还需要构造出解的策略或过程。
9. strings: 这个标签用于字符串相关的课题，如字符串匹配、操作等。

10. combinatorics: 组合数学问题，主要涉及到计数，排列，组合等问题。
11. number theory：可能涉及到一些主题。质数与因数：题目可能要求找到一个数的所有因数，判断一个数是否为质数，或者计算一个区间内的质数数量等。
    最大公约数与最小公倍数：题目可能要求计算两个数的最大公约数（GCD）或最小公倍数（LCM），或者需要利用最大公约数解决其他问题。
    同余与模运算：题目可能要求计算整数除以一个模数的余数，或者判断两个数是否是同余的，也可能涉及到欧拉定理、费马小定理等相关知识。
    数字运算：题目可能涉及到对整数进行各种运算，比如数字和、数字位数之和、数字重排等。
    整数序列：题目可能涉及到寻找某种规律的整数序列，比如斐波那契数列、卡塔兰数列等。
12. two pointers: 双指针法通常用于解决一些需要通过对数组或链表进行两个位置的遍历、对比或查找的问题。例如，在一个有序数组中查找两个数之和等于给定目标值的问题，可以使用两个指针从数组的两端向中间靠拢进行查找。这样可以减少不必要的遍历和比较。

三、Considerations 编程注意事项

2022-2023 计算概论B，TA: 李畅 戴舒羽，2022.12.13

debug通用方法

•分析报错提示

•Print中间结果，检查是否符合手动计算/推理的预期

•也可以打断点

•构造多组不同于样例的数据投入检查，最好有一些corner case

常见情况（1/2）

•各种报错和解决方案：

1. CE 2) WA 3) TLE 4) MLE 5) RTE 6) PE……

•RTE: 很可能是数组越界/除0

•TLE/MLE:

1. 程序错误，e.g.递归边界忘记设置导致一直递归，数组开得过大
2. 很大把握程序是正确的：时间复杂度/空间复杂度过高

   Solution:

   a.尝试使用更高效的数据结构 e.g. list换成dict

   b.尝试使用更高效的算法 e.g. 进行复杂度分析/设计dp/greedy etc.

   c.时间空间复杂度的平衡：用时间换空间，空间换时间

常见情况（2/2）

•不要忘记初始化变量和内存

•一道题有多组数据，不同组别之间不能相互影响

e.g.每次循环之后涉及一些变量的刷新

•dfs回溯时候要记得清空一些状态

•list的运用中，最好不要一边改一边遍历

•注意数据类型：排序时字符串按照字典序，整数型按照数的大小

•Python 里面的格式(tab)能决定一些逻辑

•复制时候浅拷贝深拷贝不分

心态调整

•考前发现环境炸了

•上来就不会做了

•认为不太难的题目怎么都AC不了

•发现考题难度激增，毫无思路

•平时没会的题型被出题助教踩了个遍

•考试过程中忍不住刷榜(理智刷榜!)

•同考场的大佬提前AC走人了

四、implementation 模拟

模拟题是一类“题目怎么说，就怎么做”的题目。这类题目通常不涉及复杂的算法，而是要求根据题目的描述进行代码的编写。考查的是代码的逻辑性和正确性，确保每一步都符合题目的要求。

下面是对这类题目的一些详细解释和示例。

02746: 约瑟夫问题

implementation, http://cs101.openjudge.cn/practice/02746

约瑟夫问题：有 n 只猴子，按顺时针方向围成一圈选大王（编号从 1 到 n ），从第 1 号开始报数，一直数到m，数到m的猴子退出圈外，剩下的猴子再接着从1开始报数。就这样，直到圈内只剩下一只猴子时，这个猴子就是猴王，编程求输入 n ， m后，输出最后猴王的编号。

> 模拟的方法直接按照题目描述的步骤进行操作，逐步移除报数到 (m) 的猴子，直到只剩下一个猴子为止。
>
> **模拟解法**
>
> 1. 初始化：
>   - 创建一个包含所有猴子编号的列表。
>   - 设置起始位置为 0（即第 1 号猴子）。
> 2. 模拟过程：
>   - 每次从当前起始位置开始报数，数到 (m) 的猴子出圈。
>   - `pop` 操作会自动调整列表的索引，因此不需要显式地重置 `pos`。
> 3. 输出结果：
>   - 输出最后剩下的猴子的编号。
>
> **代码实现**
>
> 以下是使用模拟方法解决约瑟夫问题的代码实现：

```python
def josephus(n, m):
    if n == 0:
        return 0

    monkeys = list(range(1, n + 1))  # 猴子的编号列表
    pos = 0  # 当前起始位置

    while len(monkeys) > 1:
        # 计算当前出圈的猴子的位置
        pos = (pos + m - 1) % len(monkeys)
        # 移除出圈的猴子
        monkeys.pop(pos)

    return monkeys[0]  # 返回最后剩下的猴子的编号
```

```
15
16  while True:
17      n, m = map(int, input().split())
18      if n + m == 0:
19          break
20      result = josephus(n, m)
21      print(result)
```

代码解释

1. 初始化：
   - `monkeys = list(range(1, n + 1))`：创建一个包含所有猴子编号的列表。
   - `pos = 0`：设置起始位置为 0（即第 1 号猴子）。
2. 模拟过程：
   - `pos = (pos + m - 1) % len(monkeys)`：计算当前出圈的猴子的位置。这里减1是因为我们从1开始报数，而列表索引是从0开始的。
3. 输出结果：
   - `print(result)`：输出最后剩下的猴子的编号。

03253:约瑟夫问题No.2

http://cs101.openjudge.cn/practice/03253/

n 个小孩围坐成一圈，并按顺时针编号为1,2,...,n，从编号为 p 的小孩顺时针依次报数，由1报到m ，当报到 m 时，该小孩从圈中出去，然后下一个再从1报数，当报到 m 时再出去。如此反复，直至所有的小孩都从圈中出去。请按出去的先后顺序输出小孩的编号。

直接模拟这个过程，使用一个列表来表示小孩的编号，每次报数时移除对应的小孩。

```
1   def josephus(n, p, m):
2   if n == 0:
3     return []
4
5   kids = list(range(1, n + 1))  # 孩子们的编号列表
6   pos = p - 1  # 将起始位置调整为0-based索引
7   out_order = []  # 记录出圈的顺序
8
9   while kids:
10    pos = (pos + m - 1) % len(kids)  # 计算当前出圈的孩子的位置
11    out_order.append(kids.pop(pos))  # 将出圈的孩子添加到结果列表中
12
13  return out_order
14
15  while True:
```

```
n, p, m = map(int, input().split())
if n + p + m == 0:
  break
result = josephus(n, p, m)
print(','.join(map(str, result)))
```

代码解释

1. 初始化：
   - `kids = list(range(1, n + 1))`：创建一个包含所有孩子编号的列表。
   - `pos = p - 1`：将起始位置调整为0-based索引。
   - `out_order = []`：用于记录出圈的顺序。
2. 模拟过程：
   - `pos = (pos + m - 1) % len(kids)`：计算当前出圈的孩子的位置。这里减1是因为我们从1开始报数，而列表索引是从0开始的。
   - `out_order.append(kids.pop(pos))`：将出圈的孩子添加到结果列表中，并从孩子列表中移除。
3. 输出结果：
   - `print(','.join(map(str, result)))`：将结果列表中的编号用逗号连接成字符串并输出。

其他模拟题的例子

1. 字符串操作：如字符串反转、字符串替换等。
2. 数组操作：如数组排序、数组旋转等。

五、Binary 查找

查找操作是编程中的基本技能，根据数据集的大小和结构选择合适的查找方法可以显著提高效率。线性查找适用于较小或无序的数据集，而二分查找适用于较大的有序数据集。

我发现二分查找容易理解，但是细节部分不容易写对（while的条件是<=，还是<；折半后是mid+1，mid-1，还是mid）。

> **常见的查找方法**
>
> 1. 线性查找（Linear Search）：
>   - 适用范围：适用于较小的数据集或无序的数据集。
>   - 原理：逐个检查数据集中的每个元素，直到找到满足条件的元素或遍历完所有元素。
>   - 时间复杂度：O(n)，其中 n 是数据集的大小。
> 2. 二分查找（Binary Search）：
>   - 适用范围：适用于有序的数据集。
>   - 原理：通过将数据集分成两半，逐步缩小查找范围，直到找到满足条件的元素或确定不存在。
>   - 时间复杂度：O(log n)，其中 n 是数据集的大小。
>
> **示例代码**
>
> 线性查找

```python
def linear_search(arr, target):
    for i, element in enumerate(arr):
        if element == target:
            return i  # 返回目标元素的索引
    return -1  # 如果未找到目标元素，返回 -1

# 示例
arr = [3, 5, 2, 8, 1, 9, 4]
target = 8
result = linear_search(arr, target)
print(f"Target {target} found at index {result}")
# Target 8 found at index 3
```

> 二分查找

```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid  # 返回目标元素的索引
        elif arr[mid] < target:
            left = mid + 1
        else:
```

```
            right = mid - 1

    return -1  # 如果未找到目标元素，返回 -1

# 示例
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
target = 8
result = binary_search(arr, target)
print(f"Target {target} found at index {result}")
# Target 8 found at index 7
```

详细步骤

线性查找

1. 初始化：
    - 遍历数据集中的每个元素。
2. 查找过程：
    - 逐个检查每个元素是否等于目标元素。
    - 如果找到目标元素，返回其索引。
    - 如果遍历完所有元素仍未找到目标元素，返回 -1。

二分查找

1. 初始化：
    - 设置左边界 `left` 为 0，右边界 `right` 为数据集的最后一个索引。
2. 查找过程：
    - 计算中间位置 `mid`。
    - 如果中间位置的元素等于目标元素，返回其索引。
    - 如果中间位置的元素小于目标元素，调整左边界 `left` 为 `mid + 1`。
    - 如果中间位置的元素大于目标元素，调整右边界 `right` 为 `mid - 1`。
    - 重复上述步骤，直到找到目标元素或左边界超过右边界。
3. 未找到目标元素：
    - 如果左边界超过右边界，返回 -1。

参考 bisect 源码的二分查找写法，

https://github.com/python/cpython/blob/main/Lib/bisect.py

当然可以！下面是一个基于 `bisect_left` 函数的实现，并提供一个详细的二分查找样例。

二分查找实现

```
def bisect_left(a, x, lo=0, hi=None, *, key=None):
```

```
 """Return the index where to insert item x in list a, assuming a is sorted.

 The return value i is such that all e in a[:i] have e < x, and all e in
 a[i:] have e >= x.  So if x already appears in the list, a.insert(i, x) will
 insert just before the leftmost x already there.

 Optional args lo (default 0) and hi (default len(a)) bound the
 slice of a to be searched.

 A custom key function can be supplied to customize the sort order.
 """

 if lo < 0:
     raise ValueError('lo must be non-negative')
 if hi is None:
     hi = len(a)
 # Note, the comparison uses "<" to match the
 # __lt__() logic in list.sort() and in heapq.
 if key is None:
     while lo < hi:
         mid = (lo + hi) // 2
         if a[mid] < x:
             lo = mid + 1
         else:
             hi = mid
 else:
     while lo < hi:
         mid = (lo + hi) // 2
         if key(a[mid]) < x:
             lo = mid + 1
         else:
             hi = mid
 return lo

# 二分查找函数
def binary_search(arr, target):
 index = bisect_left(arr, target)
 if index != len(arr) and arr[index] == target:
     return index  # 返回目标值的索引
 else:
     return -1  # 如果未找到目标值，返回 -1

# 示例
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
target = 8
result = binary_search(arr, target)
print(f"Target {target} found at index {result}")
```

详细步骤

1. 定义 `bisect_left` 函数：

- **参数**：
  - `a`：已排序的列表。
  - `x`：要查找的目标值。
  - `lo`：搜索范围的起始索引，默认为 0。
  - `hi`：搜索范围的结束索引，默认为 `len(a)`。
  - `key`：可选的键函数，用于自定义排序顺序。
- **逻辑**：
  - 检查 `lo` 是否非负。
  - 如果 `hi` 为 `None`，则设置 `hi` 为 `len(a)`。
  - 使用二分查找算法找到目标值 `x` 应该插入的位置。
  - 如果 `key` 为 `None`，直接比较 `a[mid]` 和 `x`。
  - 如果 `key` 不为 `None`，比较 `key(a[mid])` 和 `x`。

2. **定义 `binary_search` 函数**：
   - 使用 `bisect_left` 找到目标值在已排序列表中第一次出现的位置。
   - 检查目标值是否存在于列表中：
     - 如果 `index` 不等于列表的长度且 `arr[index]` 等于目标值，返回 `index`。
     - 否则，返回 -1。

自定义键函数示例

假设你有一个包含元组的列表，并且你希望根据元组的第二个元素进行二分查找：

```
def binary_search_with_key(arr, target, key):
    index = bisect_left(arr, target, key=key)
    if index != len(arr) and key(arr[index]) == target:
        return index  # 返回目标值的索引
    else:
        return -1  # 如果未找到目标值，返回 -1

# 示例
arr = [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
target = 'c'
result = binary_search_with_key(arr, target, key=lambda x: x[1])
print(f"Target {target} found at index {result}")
```

- **输入**：

```
arr = [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
target = 'c'
```

- **输出**：

```
Target c found at index 2
```

总结

二分查找是一种高效的查找算法，适用于已排序的数据集。你可以使用 `bisect` 模块中的 `bisect_left` 函数来快速实现二分查找，也可以手动实现以学习算法的细节。

六、Common 知识点

浅拷贝

在 Python 中声明二维数组时，需要注意避免浅拷贝问题。浅拷贝会导致所有行引用同一个列表，从而在修改一个元素时影响到其他行。为了避免这个问题，可以使用嵌套的列表推导式或循环来创建独立的子列表

> **避免浅拷贝问题**：
>
> - 如果使用 `[[0] * n] * m` 这种方式创建二维数组，会导致所有行引用同一个列表，修改一个元素会影响所有行。例如：

```
matrix = [[0] * n] * m
matrix[0][0] = 1
print(matrix)
# 输出：[[1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0]]
```

> - 因此，推荐使用列表推导式或嵌套循环来创建独立的子列表。
>
> **使用列表推导式**

```
# 创建一个 m x n 的二维数组，所有元素初始化为 0
m = 3
n = 4
matrix = [[0 for _ in range(n)] for _ in range(m)]
print(matrix)
# 输出：[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

> **使用嵌套循环**

```
# 创建一个 m x n 的二维数组，所有元素初始化为 0
m = 3
n = 4
matrix = []
for i in range(m):
    row = []
    for j in range(n):
        row.append(0)
    matrix.append(row)
print(matrix)
# 输出：[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

解释

> 1. **使用列表推导式**：
>   - `[[0 for _ in range(n)] for _ in range(m)]`：
>   - 外层的 `for _ in range(m)` 生成 `m` 个子列表。
>   - 内层的 `for _ in range(n)` 生成每个子列表中的 `n` 个元素，每个元素初始化为 `0`。

- 这种方法简洁且易于理解，适合快速创建多维数组。

2. **使用嵌套循环**：
   - 外层的 `for i in range(m)` 循环 `m` 次，每次创建一个新的子列表 `row`。
   - 内层的 `for j in range(n)` 循环 `n` 次，每次向 `row` 中添加一个 `0`。
   - 最后将 `row` 添加到 `matrix` 中。
   - 这种方法虽然代码稍长，但逻辑清晰，适合需要更复杂初始化的情况。

请问一下完美立方为什么从a开始循环会少答案?

浮点数精度问题

python开三次方根的误差特别大

确实，处理浮点数时很容易因为精度问题而出现错误。因此，如果可以通过乘法实现的话，应尽量避免使用除法。这样可以减少由于浮点数运算带来的误差，提高计算的准确性。

无穷大

浮点数无穷大

在 Python 中，处理无穷大的概念时，浮点数和整数的行为有所不同。

浮点数无穷大

对于浮点数，Python 支持表示正无穷大和负无穷大。可以使用 `float("inf")` 表示正无穷大，`float("-inf")` 表示负无穷大。例如：

```python
positive_infinity = float("inf")
negative_infinity = float("-inf")

print(positive_infinity)  # 输出: inf
print(negative_infinity)  # 输出: -inf

# 使用 math.inf
import math

positive_infinity_float = math.inf
negative_infinity_float = -math.inf
print(positive_infinity_float)  # 输出: inf
print(negative_infinity_float)  # 输出: -inf
```

整数无穷大

> 对于整数，Python 并没有直接支持表示无穷大的整数。整数类型 `int` 在 Python 中是没有固定大小限制的，可以表示任意大的整数，但仍然不是无穷大。如果你尝试将 `"inf"` 转换为整数，会引发一个 `ValueError`：
>
> ```
> try:
>  int("inf")
> except ValueError as e:
>  print(e)  # 输出: invalid literal for int() with base 10: 'inf'
> ```

整数无穷大

> 在 Python 中，由于整数类型 `int` 没有固定的大小限制，理论上可以表示任意大的整数。但是，Python 并没有直接提供表示整数无穷大的语法或内置常量。如果你需要在程序中表示整数无穷大，可以考虑以下几种方法：
>
> **使用一个非常大的整数**
>
> 你可以选择一个足够大的整数来表示无穷大。这种方法简单直接，但需要注意选择的数值要足够大，以避免在实际计算中被误认为是一个普通的有限值。
>
> ```
> INFINITY = 10**1000  # 选择一个非常大的整数
> NEGATIVE_INFINITY = -10**1000  # 选择一个非常小的整数
> ```

常用函数

问一个初学者的问题:为什么在列表的方法中，有的方法可以连同列表本身作为一个“变量”，比如print(list1.pop())可以输出list1中最后一个元素，而有的方法却不行，比如print(list1.sort())会输出None（而不是排序后的list1）?区别这两种方法是只能靠记忆吗?

> 这是一个很好的问题，涉及到 Python 中列表方法的行为差异。实际上，这些方法之间的区别并不是只能靠记忆，而是可以通过理解它们的工作方式来掌握。
>
> 1. **返回值**：
>   - 一些列表方法，如 `pop()`，执行操作后会返回一个值。`pop()` 方法从列表中移除并返回指定位置的元素，默认情况下是最后一个元素。因此，当你调用 `print(list1.pop())` 时，你会看到被移除的元素被打印出来。
>   - 另一些方法，如 `sort()`，则是在原地对列表进行修改，并不返回任何值（或者更准确地说，返回 `None`）。这是因为 `sort()` 方法直接改变了列表本身的顺序，没有必要再返回整个列表。所以当你尝试 `print(list1.sort())` 时，实际上打印的是 `None`。
> 2. **原地操作 vs. 返回新对象**：

- 像 `sort()` 这样的方法是对列表进行原地修改的，即它们改变的是原始列表，而不是创建一个新的已排序列表。
- 相反，如果一个方法返回了一个新的对象，比如 `sorted(list1)`，它会返回一个新的排序后的列表，而原始列表 `list1` 不会发生变化。

3. **如何判断**：
   - 要判断一个方法是否返回值，最好的办法是查阅官方文档或使用帮助功能（如 `help(list.sort)`）。文档通常会明确指出一个方法的行为，包括它是否修改了列表以及是否返回了值。
   - 经验上，很多修改数据结构的方法（如 `append()`, `remove()`, `reverse()` 等）都是原地操作，而那些用于查询或生成新数据的方法（如 `count()`, `index()`, `copy()` 等）则会返回值。

需要熟悉常用函数的返回值。常用的函数有限的，刷30个CF题目基本就都知道了。例如：input,split map,join,enumerate, print,append,sort,remove, index,find,.....

想用的时候先在交互式窗口试一下，help(函数名)。

对于在Codeforces上写Python代码的程序，以下是一些常用的函数和功能：

1. 输入和输出：
   - `input()`：从标准输入（键盘）获取一行输入
   - `print()`：输出一个或多个表达式
   - `map(func, *iterables)`：将函数应用于传入的每个可迭代对象的各个元素
2. 数学计算：
   - `abs()`：计算绝对值
   - `max()` 和 `min()`：分别获取最大值和最小值
   - `sum()`：计算所有元素的总和
   - `pow(x, y)`：计算x的y次方
   - `math.floor()`：向下取整
   - `math.ceil()`：向上取整
3. 列表操作：
   - `len()`：返回列表长度
   - `append()`：在列表末尾添加新的对象
   - `extend()`：在列表末尾一次性追加另一个序列中的多个值
   - `insert()`：将对象插入列表
   - `sort()`：对列表进行排序
   - `list()`：生成列表
4. 字符串操作：
   - `str.replace()`：字符串替换

- `str.split()`：字符串分割
- `str.strip()`：移除字符串头尾指定的字符（默认为空格或换行符）或字符序列
- `str.join()`：用于将序列中的元素以指定的字符连接生成一个新的字符串

5. 组合数据类型操作：
   - `set()`：创建一个无序不重复元素集，可进行关系测试，删除重复数据，还可以计算交集、差集、并集等
   - `dict()`：创建一个字典
6. 控制流：
   - `if..elif..else`：条件判断
   - `for...in`：循环
   - `while`：循环
7. 其他有用的内置函数：
   - `range()`：生成一个数字序列
   - `enumerate()`：为迭代器增加索引标签，常用于获取列表的元素及其索引

以上是一些基础的Python函数，另外根据题目需要，可能需要用到其他Python标准库如`itertools`（作用于迭代器的函数），`collections`（高性能容器datatypes），`heapq`（提供堆数据结构的函数）等。

七、Optimize and improve算法

Python 是一种高级编程语言，具有丰富的库和简洁的语法。然而，在处理大规模数据或高性能要求的应用时，优化代码是非常重要的。以下是一些常见的 Python 优化的技巧：

1. **使用内置函数和库**

Python 内置了许多高效的数据结构和函数，尽量利用这些内置功能可以提高性能。

示例：

- **列表推导式**：比传统的 `for` 循环更快。

```python
# 传统方式
squares = []
for x in range(10):
    squares.append(x ** 2)

# 列表推导式
squares = [x ** 2 for x in range(10)]
```

- **`map()` 和 `filter()`**：比列表推导式稍慢，但在某些情况下更清晰。

```python
# 传统方式
numbers = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x ** 2, numbers))

# 列表推导式
squares = [x ** 2 for x in numbers]
```

2. **使用生成器**

生成器可以在需要时生成值，而不是一次性生成所有值，从而节省内存。

示例：

```python
# 生成器表达式
squares = (x ** 2 for x in range(10))

# 生成器函数
def generate_squares(n):
    for x in range(n):
        yield x ** 2

# 使用生成器
for square in generate_squares(10):
    print(square)
```

3. 使用 `set` 和 `dict` **进行成员测试**

`set` 和 `dict` 的成员测试时间复杂度为 O(1)，而 `list` 和 `tuple` 的成员测试时间复杂度为 O(n)。

示例：

```python
# 使用 list
numbers = list(range(1000000))
print(999999 in numbers)  # 较慢

# 使用 set
numbers_set = set(numbers)
print(999999 in numbers_set)  # 快速
```

4. **避免全局变量**

访问全局变量比访问局部变量慢，尽量将变量放在局部作用域内。

示例：

```python
global_var = 1000000

def function_with_global():
    global global_var
    for _ in range(1000000):
        global_var += 1

def function_with_local():
    local_var = 1000000
    for _ in range(1000000):
        local_var += 1

# 测量时间
import time

start = time.time()
function_with_global()
print("Global variable:", time.time() - start)

start = time.time()
function_with_local()
print("Local variable:", time.time() - start)
```

5. 使用 `functools.lru_cache` 缓存结果

对于耗时的函数调用，可以使用 `functools.lru_cache` 来缓存结果，避免重复计算。

示例：

```python
from functools import lru_cache

@lru_cache(maxsize=128)
def fibonacci(n):
    if n < 2:
```

```
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

# 测量时间
import time

start = time.time()
print(fibonacci(30))
print("Time taken:", time.time() - start)
```

6. 使用 `numpy` 和 `pandas` 处理数值数据

`numpy` 和 `pandas` 是专门为数值计算优化的库，可以显著提高性能。

示例：

```
import numpy as np

# 传统方式
a = [i for i in range(1000000)]
b = [i * 2 for i in a]

# 使用 numpy
a_np = np.arange(1000000)
b_np = a_np * 2

# 测量时间
import time

start = time.time()
c = [x + y for x, y in zip(a, b)]
print("List time:", time.time() - start)

start = time.time()
c_np = a_np + b_np
print("NumPy time:", time.time() - start)
```

7. 使用多线程或多进程

对于 CPU 密集型任务，可以使用多进程；对于 I/O 密集型任务，可以使用多线程。

示例：

```
import concurrent.futures

def process_data(data):
    # 模拟耗时操作
    return data * 2

data_list = list(range(1000000))

# 单线程
```

```python
start = time.time()
results = [process_data(data) for data in data_list]
print("Single thread time:", time.time() - start)

# 多线程
with concurrent.futures.ThreadPoolExecutor() as executor:
    start = time.time()
    results = list(executor.map(process_data, data_list))
    print("Multi-thread time:", time.time() - start)

# 多进程
with concurrent.futures.ProcessPoolExecutor() as executor:
    start = time.time()
    results = list(executor.map(process_data, data_list))
    print("Multi-process time:", time.time() - start)
```

~~8. 使用 `Cython` 或 `Numba` 进行 JIT 编译~~

~~`Cython` 和 `Numba` 可以将 Python 代码编译成 C 代码，从而提高性能。~~

~~示例：~~

```python
# 使用 Numba
import numba

@numba.jit(nopython=True)
def sum_of_squares(n):
    result = 0
    for i in range(n):
        result += i ** 2
    return result

# 测量时间
import time

start = time.time()
result = sum_of_squares(10000000)
print("Numba time:", time.time() - start)
```

~~9. 使用 `PyPy` 作为解释器~~

~~`PyPy` 是一个兼容 Python 的解释器，使用 JIT 编译技术，可以显著提高某些类型程序的性能。~~

~~10. 代码分析和优化~~

~~使用 `cProfile` 和 `line_profiler` 等工具进行性能分析，找出瓶颈并进行优化。~~

~~示例：~~

```python
import cProfile
import pstats

```

```python
def my_function():
    # 模拟耗时操作
    for _ in range(1000000):
        pass

cProfile.run('my_function()', 'profile_stats')

# 分析结果
with open('profile_stats', 'r') as f:
    stats = pstats.Stats(f)
    stats.sort_stats('cumulative').print_stats(10)
```

通过这些优化技巧，可以显著提高 Python 代码的性能。选择合适的优化方法取决于具体的使用场景和需求。

230B. T-primes

binary search/implementation/math/number theory, 1300, http://codeforces.com/problemset/problem/230/B

We know that prime numbers are positive integers that have exactly two distinct positive divisors. Similarly, we'll call a positive integer *t* T-prime, if *t* has exactly three distinct positive divisors.

You are given an array of *n* positive integers. For each of them determine whether it is T-prime or not.

Input

The first line contains a single positive integer, *n* ($1 \le n \le 10^5$), showing how many numbers are in the array. The next line contains *n* space-separated integers x_i ($1 \le x_i \le 10^{12}$).

Please, do not use the %lld specifier to read or write 64-bit integers in C++. It is advised to use the cin, cout streams or the %I64d specifier.

Output

Print *n* lines: the *i*-th line should contain "YES" (without the quotes), if number x_i is T-prime, and "NO" (without the quotes), if it isn't.

Examples

input

```
3
4 5 6
```

output

```
YES
NO
NO
```

Note

The given test has three numbers. The first number 4 has exactly three divisors — 1, 2 and 4, thus the answer for this number is "YES". The second number 5 has two divisors (1 and 5), and the third number 6 has four divisors (1, 2, 3, 6), hence the answer for them is "NO".

与这个题目类似，都是找平方数优化。01218:THE DRUNK JAILER，http://cs101.openjudge.cn/practice/01218/

这个题目实际上是DP思路。

数论是有趣和优美的数学分支。欧几里得对于素数无穷性的证明在今天看来仍和两千年前一样清晰和优雅。长久以来，计算机都被用来辅助数论研究，有很多精妙的算法能够帮上忙。

求解素数的三种方法，包括：试除法（trial division）、埃氏筛（Sieve of Eratosthenes）、欧拉筛（Sieve of Euler，线性法），https://blog.dotcpp.com/a/69737

优化的埃式筛法、或者欧拉筛，可以到 1秒以下。

Python3, Accepted, 748ms。用到了埃式筛法，内置函数math.sqrt, math.isqrt，set，第8行循环从i*i开始，使用sys.stdin.read() 一次性读取所有输入，缓存一次性输出。

```python
import math

def sieve(limit):
    is_prime = [True] * (limit + 1)
    is_prime[0] = is_prime[1] = False
    for i in range(2, int(math.sqrt(limit)) + 1):
        if is_prime[i]:
            for j in range(i * i, limit + 1, i):
                is_prime[j] = False
    return [i for i in range(limit + 1) if is_prime[i]]

def is_t_prime(x, primes_set):
    if x < 2:
        return False
    root = int(math.isqrt(x))
    return root * root == x and root in primes_set

def main():
    import sys
    input = sys.stdin.read
    data = input().split()

    n = int(data[0])
    numbers = list(map(int, data[1:n+1]))
```

```
25
26      limit = 10**6
27      primes = sieve(limit)
28      primes_set = set(primes)
29
30      results = []
31      for number in numbers:
32          if is_t_prime(number, primes_set):
33              results.append("YES")
34          else:
35              results.append("NO")
36
37      print("\n".join(results))
38
39  if __name__ == "__main__":
40      main()
```

可以进一步优化，直接用is_prim列表。Python3, Accepted, 654ms。

```
1   import math
2
3   def sieve(limit):
4       is_prime = [True] * (limit + 1)
5       is_prime[0] = is_prime[1] = False
6       for i in range(2, int(math.sqrt(limit)) + 1):
7           if is_prime[i]:
8               for j in range(i * i, limit + 1, i):
9                   is_prime[j] = False
10      #return [i for i in range(limit + 1) if is_prime[i]]
11      return is_prime
12
13  #def is_t_prime(x, primes_set):
14  def is_t_prime(x, primes_list):
15      if x < 2:
16          return False
17      root = int(math.isqrt(x))
18      #return root * root == x and root in primes_set
19      return root * root == x and primes_list[root]
20
21  def main():
22      import sys
23      input = sys.stdin.read
24      data = input().split()
25
26      n = int(data[0])
27      numbers = list(map(int, data[1:n+1]))
28
29      limit = 10**6
30      primes = sieve(limit)
31      #primes_set = set(primes)
```

```
32
33      results = []
34      for number in numbers:
35          #if is_t_prime(number, primes_set):
36          if is_t_prime(number, primes):
37              results.append("YES")
38          else:
39              results.append("NO")
40
41      print("\n".join(results))
42
43  if __name__ == "__main__":
44      main()
```

参考

通义千问，https://tongyi.aliyun.com